

**AD-A240 543****MENTATION PAGE**Form Approved  
OPM No. 0704-0188**2**

average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington, DC 20543, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget.

|   |  |   |   |
|---|--|---|---|
| 1. AGENCY USE ONLY (Leave Blank)  |  | 2. REPORT DATE  | 3. REPORT TYPE AND DATES COVERED<br>Final: 01 Aug 1991 to 01 Jun 1993 |
| 4. TITLE AND SUBTITLE<br>Verdix Corporation, VADS VAX/VMS=> Intel 386/WITEK 3167, VMS 5.2, VAda-110-03315, Version 6.0, MicroVAX 3100 (Host) to iSBC 386/116 (using a WEITEK 3167 fpu)(Target), 901129W1.11094  |  | 5. FUNDING NUMBERS  |   |
| 6. AUTHOR(S)<br>Wright-Patterson AFB, Dayton, OH<br>USA   |  | 8. PERFORMING ORGANIZATION<br>REPORT NUMBER<br>AVF-VSR-404-0891   |   |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>Ada Validation Facility, Language Control Facility ASD/SCEL<br>Bldg. 676, Rm 135<br>Wright-Patterson AFB<br>Dayton, OH 45433  |  | 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)<br>Ada Joint Program Office<br>United States Department of Defense<br>Pentagon, Rm 3E114<br>Washington, D.C. 20301-3081 |   |
| 10. SPONSORING/MONITORING AGENCY<br>REPORT NUMBER   |  |   |   |
| 11. SUPPLEMENTARY NOTES   |  |   |   |
| 12a. DISTRIBUTION/AVAILABILITY STATEMENT<br>Approved for public release; distribution unlimited.  |  | 12b. DISTRIBUTION CODE  |   |
| 13. ABSTRACT (Maximum 200 words)<br>Verdix Corporation, VADS VAX/VMS=> Intel 386/WEITEK 3167, VMS 5.2, VAda-110-03315, Version 6.0, Wright-Patterson AFB, MicroVAX 3100, VMS 5.2 (Host) to Intel iSBC 386/116 (using a WEITEK 3167 fpu), (bare machine)(Target), ACVC 1.11. |  |   |   |
| 14. SUBJECT TERMS<br>Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.  |  | 15. NUMBER OF PAGES   |   |
| 17. SECURITY CLASSIFICATION<br>OF REPORT<br>UNCLASSIFIED  |  | 18. SECURITY CLASSIFICATION<br>UNCLASSIFIED   |   |
| 19. SECURITY CLASSIFICATION<br>OF ABSTRACT<br>UNCLASSIFIED  |  | 20. LIMITATION OF ABSTRACT  |   |
| 16. PRICE CODE  |  |   |   |

**DTIC**  
**SELECTED**  
**SEP 19 1991**  
**S D****91-11071**

AVF Control Number: AVF-VSR-404-0891

1 August 1991

90-09-25-VRX

Ada COMPILER

VALIDATION SUMMARY REPORT:

Certificate Number: 901129W1.11094

Verdix Corporation

VADS VAX/VMS => Intel 386/WEITEK 3167, VMS 5.2, VAdA-110-03315, Version 6.0

MicroVAX 3100 => iSBC 386/116 (using a WEITEK 3167 fpu)

Prepared By:

Ada Validation Facility

ASD/SCEL

Wright-Patterson AFB OH 45433-6503

|                    |  |
|--------------------|--|
| Accession For      |  |
| NTIS CRA&I         | <input checked="checked" type="checkbox"/> |
| DTIC TAB           | <input type="checkbox"/>                   |
| Unannounced        | <input type="checkbox"/>                   |
| Justification      |  |
| By                 |  |
| Distribution       |  |
| Availability Codes |  |
| Dist               | Avail a b/c or Special                     |
| A-1                |  |



### Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 29 November 1990.

Compiler Name and Version: VADS VAX/VMS => Intel 386/WEITEK 3167,  
VMS 5.2, VAda-110-03315, Version 6.0

Host Computer System: MicroVAX 3100, VMS 5.2

Target Computer System: Intel iSBC 386/116 (using a WEITEK 3167 fpu),  
(bare machine)

Customer Agreement Number: 90-09-25-VRX

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901129W1.11094 is awarded to Verdix Corporation. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.



Ada Validation Facility  
Steven P. Wilson  
Technical Director  
ASD/SCEL  
Wright-Patterson AFB OH 45433-6503



for Ada Validation Organization  
Director, Computer & Software Engineering Division  
Institute for Defense Analyses  
Alexandria VA 22311



on Ada Joint Program Office  
Dr. John Solomond, Director  
Department of Defense  
Washington DC 20301

# DECLARATION OF CONFORMANCE

Customer:

Verdix Corporation

Ada Validation Facility:

ASD/SCEL Wright Patterson AFB, OH 45433-6503

Ada Compiler Validation Capability (ACVC)

Version: 1.11

## Ada Implementation

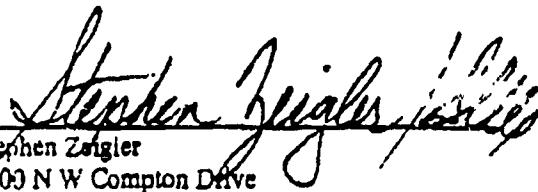
Compiler Name: VADS VAX/VMS=>Intel 386, WEITEK 3167, VMS 5.2, VAda-110-03315, Version: 6.0

Host Computer System: MicroVAX 3100, VMS 5.2

Target Computer System: iSBC 386/116 (using a WEITEK 3167 fpu), (bare machine)

## Declaration

I, the undersigned, representing Verdix Corporation, declare that Verdix Corporation has no knowledge of deliberate deviations from the Ada Language Standard ANS/MIL-STD-1815A in the implementation listed in this declaration. I declare that the Verdix Corporation is the owner of the above implementation and the certificates shall be awarded in the name of Verdix Corporation.

  
Stephen Zeigler  
1600 N W Compton Drive  
Suite 357  
Beaverton Oregon 97006

Date:

2/11/91

## CHAPTER 2

### IMPLEMENTATION DEPENDENCIES

#### 2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AV0. The rationale for withdrawing each test is available from either the AV0 or the AVF. The publication date for this list of withdrawn tests is 12 October 1990.

|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
| E28005C | B28006C | C34006D | B41308B | C43004A | C45114A |
| C45346A | C45612B | C45651A | C46022A | B49008A | A74006A |
| C74308A | B83022B | B83022H | B83025B | B83025D | B83026B |
| B85001L | C83026A | C83041A | C97116A | C98003B | BA2011A |
| CB7001A | CB7001B | CB7004A | CC1223A | BC1226A | CC1226B |
| BC3009B | BD1B02B | BD1B06A | AD1B08A | BD2A02A | CD2A21E |
| CD2A23E | CD2A32A | CD2A41A | CD2A41E | CD2A87A | CD2Bi5C |
| BD3006A | BD4008A | CD4022A | CD4022D | CD4024B | CD4024C |
| CD4024D | CD4031A | CD4051D | CD5111A | CD7004C | ED7005D |
| CD7005E | AD7006A | CD7006E | AD7201A | AD7201E | CD7204B |
| BD8002A | BD8004C | CD9005A | CD9005B | CDA201E | CE2107I |
| CE2117A | CE2117B | CE2119B | CE2205B | CE2405A | CE3111C |
| CE3118A | CE3411B | CE3412B | CE3607B | CE3607C | CE3607D |
| CE3812A | CE3814A | CE3902B |         |         |         |

#### 2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

## IMPLEMENTATION DEPENDENCIES

The following 201 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

|                       |                       |
|-----------------------|-----------------------|
| C24113L..Y (14 tests) | C35705L..Y (14 tests) |
| C35706L..Y (14 tests) | C35707L..Y (14 tests) |
| C35708L..Y (14 tests) | C35802L..Z (15 tests) |
| C45241L..Y (14 tests) | C45321L..Y (14 tests) |
| C45421L..Y (14 tests) | C45521L..Z (15 tests) |
| C45524L..Z (15 tests) | C45621L..Z (15 tests) |
| C45641L..Y (14 tests) | C46012L..Z (15 tests) |

The following 21 tests check for the predefined type `LONG_INTEGER`:

|         |         |         |         |         |
|---------|---------|---------|---------|---------|
| C35404C | C45231C | C45304C | C45411C | C45412C |
| C45502C | C45503C | C45504C | C45504F | C45611C |
| C45612C | C45613C | C45614C | C45631C | C45632C |
| B52004D | C55B07A | B55B09C | B86001W | C86006C |
| CD7101F |         |         |         |         |

C35702B, C35713C, B86001U, and C86006G check for the predefined type `LONG_FLOAT`.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`.

A35801E checks that `FLOAT'FIRST..FLOAT'LAST` may be used as a range constraint in a floating-point type declaration; for this implementation that range exceeds the safe numbers and must be rejected. (See section 2.3)

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater.

C45624A..B (2 tests) check that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types; for this implementation, `MACHINE_OVERFLOW` is `TRUE`.

C86001F recompiles package `SYSTEM`, making package `TEXT_IO`, and hence package `REPORT`, obsolete. For this implementation, the package `TEXT_IO` is dependent upon package `SYSTEM`.

B86001Y checks for a predefined fixed-point type other than `DURATION`.

C96005B checks for values of type `DURATION'BASE` that are outside the range of `DURATION`. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

## IMPLEMENTATION DEPENDENCIES

The tests listed in the following table are not applicable because the given file operations are not supported for the given combination of mode and file access method:

| Test    | File Operation | Mode                 | File Access Method |
|---------|----------------|----------------------|--------------------|
| CE2102D | CREATE         | IN FILE              | SEQUENTIAL_IO      |
| CE2102E | CREATE         | OUT FILE             | SEQUENTIAL_IO      |
| CE2102F | CREATE         | INO $\bar{U}$ T FILE | DIRECT_IO          |
| CE2102I | CREATE         | IN FILE              | DIRECT_IO          |
| CE2102J | CREATE         | OUT FILE             | DIRECT_IO          |
| CE2102N | OPEN           | IN FILE              | SEQUENTIAL_IO      |
| CE2102O | RESET          | IN FILE              | SEQUENTIAL_IO      |
| CE2102P | OPEN           | OUT FILE             | SEQUENTIAL_IO      |
| CE2102Q | RESET          | OUT FILE             | SEQUENTIAL_IO      |
| CE2102R | OPEN           | INO $\bar{U}$ T FILE | DIRECT_IO          |
| CE2102S | RESET          | INO $\bar{U}$ T FILE | DIRECT_IO          |
| CE2102T | OPEN           | IN FILE              | DIRECT_IO          |
| CE2102U | RESET          | IN FILE              | DIRECT_IO          |
| CE2102V | OPEN           | OUT FILE             | DIRECT_IO          |
| CE2102W | RESET          | OUT FILE             | DIRECT_IO          |
| CE3102E | CREATE         | IN FILE              | TEXT_IO            |
| CE3102F | RESET          | Any Mode             | TEXT_IO            |
| CE3102G | DELETE         | -----                | TEXT_IO            |
| CE3102I | CREATE         | OUT FILE             | TEXT_IO            |
| CE3102J | OPEN           | IN FILE              | TEXT_IO            |
| CE3102K | OPEN           | OUT FILE             | TEXT_IO            |

CE2107A..E (5 tests), CE2107L, CE2110B, and CE2111D attempt to associate multiple internal files with the same external file for sequential files. The proper exception is raised when multiple access is attempted.

CE2107F..H (3 tests), CE2110D, and CE2111H attempt to associate multiple internal files with the same external file for direct files. The proper exception is raised when multiple access is attempted.

CE3111A..B (2 tests), CE3111D..E (2 tests), CE3114B, and CE3115A attempt to associate multiple internal files with the same external file for text files. The proper exception is raised when multiple access is attempted.

CE2203A checks that WRITE raises USE\_ERROR if the capacity of the external file is exceeded for SEQUENTIAL\_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE\_ERROR if the capacity of the external file is exceeded for DIRECT\_IO. This implementation does not restrict file capacity.

## IMPLEMENTATION DEPENDENCIES

CE3304A checks that USE\_ERROR is raised if a call to SET\_LINE\_LENGTH or SET\_PAGE\_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT\_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

### 2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 22 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests:

|         |         |         |         |         |
|---------|---------|---------|---------|---------|
| B24009A | B33301B | B38003A | B38003B | B38009A |
| B38009B | B85008G | B85008H | BC1303F | BC3005B |
| BD2B03A | BD2D03A | BD4003A |         |         |

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO; the compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. ARM 3.5.7(12)).

CD1009A, CD1009I, CD1C03A, CD2A22J, CD2A24A, and CD2A31A..C (3 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests use instantiations of the support procedure LENGTH\_CHECK, which uses Unchecked\_Conversion according to the interpretation given in AI-00590. The AVO ruled that this interpretation is not binding under ACVC 1.11; the tests are ruled to be passed if they produce Failed messages only from the instances of LENGTH\_CHECK--i.e., the allowed Report.Failed messages have the general form:

" \* CHECK ON REPRESENTATION FOR <TYPE\_ID> FAILED."



APPENDIX C  
APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is
    .....
    type INTEGER is range -2147483648 .. 2147483647;
    type SHORT_INTEGER is range -32768 .. 32767;
    type TINY_INTEGER is range -128 .. 127;
    type FLOAT is digits 15 range
        -1.79769313486232E+308 .. +1.79769313486232E+308;
    type SHORT_FLOAT is digits 6 range
        -3.40282E38.. +3.40282E38;
    type DURATION is delta 0.001 range -2147483.648 ..
        2147483.647;
    .....
end STANDARD;
```

## CHAPTER 3

### PROCESSING INFORMATION

#### 3.1 TESTING ENVIRONMENT

The host and target computer systems for the Ada implementation were connected by Ethernet. The rest of this Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Steve Hodges  
Verdix Corporation  
14130-A Sully Field Circle  
Chantilly VA 22021

For a point of contact for sales information about this Ada implementation system, see:

Steve Hodges  
Verdix Corporation  
14130-A Sully Field Circle  
Chantilly VA 22021

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

#### 3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

## PROCESSING INFORMATION

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

|  |              |
|--|--------------|
| a) Total Number of Applicable Tests                | 3797         |
| b) Total Number of Withdrawn Tests                 | 81           |
| c) Processed Inapplicable Tests                    | 91           |
| d) Non-Processed I/O Tests                         | 0            |
| e) Non-Processed Floating-Point<br>Precision Tests | 201          |
| f) Total Number of Inapplicable Tests              | 292 (c+d+e)  |
| g) Total Number of Tests for ACVC 1.11             | 4170 (a+b+f) |

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

### 3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 292 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were not loaded directly onto the host computer. The tape was loaded onto a Sun Workstation, and the tests were copied over Ethernet to the host machine.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link described above, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options.

## PROCESSING INFORMATION

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

## APPENDIX A

### MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The following macro parameters are defined in terms of the value V of \$MAX\_IN\_LEN which is the maximum input line length permitted for the tested implementation. For these parameters, Ada string expressions are given rather than the macro values themselves.

| Macro Parameter              | Macro Value                                    |
|------------------------------|--|
| \$BIG_ID1                    | (1..V-1 => 'A', V => '1')                      |
| \$BIG_ID2                    | (1..V-1 => 'A', V => '2')                      |
| \$BIG_ID3                    | (1..V/2 => 'A') & '3' &<br>(1..V-1-V/2 => 'A') |
| \$BIG_ID4                    | (1..V/2 => 'A') & '4' &<br>(1..V-1-V/2 => 'A') |
| \$BIG_INT_LIT                | (1..V-3 => '0') & "298"                        |
| \$BIG_REAL_LIT               | (1..V-5 => '0') & "690.0"                      |
| \$BIG_STRING1                | '"' & (1..V/2 => 'A') & '"'                    |
| \$BIG_STRING2                | '"' & (1..V-1-V/2 => 'A') & '1' & '"'          |
| \$BLANKS                     | (1..V-20 => ' ')                               |
| \$MAX_LEN_INT_BASED_LITERAL  | "2:" & (1..V-5 => '0') & "11:"                 |
| \$MAX_LEN_REAL_BASED_LITERAL | "16:" & (1..V-7 => '0') & "F.E:"               |
| \$MAX_STRING_LITERAL         | '"' & (1..V-2 => 'A') & '"'                    |

## MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values:

| Macro Parameter                   | Macro Value                       |
|-----------------------------------|-----------------------------------|
| \$MAX_IN_LEN                      | 499                               |
| \$ACC_SIZE                        | 32                                |
| \$ALIGNMENT                       | 4                                 |
| \$COUNT_LAST                      | 2_147_483_647                     |
| \$DEFAULT_MEM_SIZE                | 16_777_216                        |
| \$DEFAULT_STOR_UNIT               | 8                                 |
| \$DEFAULT_SYS_NAME                | I386                              |
| \$DELTA_DOC                       | 0.0000000004656612873077392578125 |
| \$ENTRY_ADDRESS                   | SYSTEM."+"(16#40#)                |
| \$ENTRY_ADDRESS1                  | SYSTEM."+"(16#80#)                |
| \$ENTRY_ADDRESS2                  | SYSTEM."+"(16#100#)               |
| \$FIELD_LAST                      | 2_147_483_647                     |
| \$FILE_TERMINATOR                 | ' '                               |
| \$FIXED_NAME                      | NO_SUCH_TYPE                      |
| \$FLOAT_NAME                      | NO_SUCH_TYPE                      |
| \$FORM_STRING                     | ""                                |
| \$FORM_STRING2                    | "CANNOT_RESTRICT_FILE_CAPACITY"   |
| \$GREATER_THAN_DURATION           | 100_000.0                         |
| \$GREATER_THAN_DURATION_BASE_LAST | 10_000_000.0                      |
| \$GREATER_THAN_FLOAT_BASE_LAST    | 1.8E+308                          |
| \$GREATER_THAN_FLOAT_SAFE_LARGE   | 5.0E+307                          |

# MACRO PARAMETERS

```

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
    9.0E+37

$HIGH_PRIORITY      99

$ILLEGAL_EXTERNAL_FILE_NAME1
    "/illegal/file_name/2}]%2102c.dat"

$ILLEGAL_EXTERNAL_FILE_NAME2
    "/illegal/file_name/CE2102C*.DAT"

$INAPPROPRIATE_LINE_LENGTH
    -1

$INAPPROPRIATE_PAGE_LENGTH
    -1

$INCLUDE_PRAGMA1    PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2    PRAGMA INCLUDE ("B28006D1.TST")

$INTEGER_FIRST      -2_147_483_648
$INTEGER_LAST        2_147_483_647
$INTEGER_LAST_PLUS_1 2_147_483_648

$INTERFACE_LANGUAGE  C

$LESS_THAN_DURATION -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
    -10_000_000.0

$LINE_TERMINATOR     ASCII.LF & ASCII.FF

$LOW_PRIORITY        0

$MACHINE_CODE_STATEMENT
    CODE_0'(OP => NOP);

$MACHINE_CODE_TYPE   CODE_0

$MANTISSA_DOC         31

$MAX_DIGITS           15

$MAX_INT              2_147_483_647
$MAX_INT_PLUS_1       2_147_483_648
$MIN_INT              -2_147_483_648

```

## MACRO PARAMETERS

|                       |                                    |
|-----------------------|------------------------------------|
| \$NAME                | TINY_INTEGER                       |
| \$NAME_LIST           | I386                               |
| \$NAME_SPECIFICATION1 | VMS386\$DKB0:[ACVC1.11.C.E]X2120A. |
| \$NAME_SPECIFICATION2 | VMS386\$DKB0:[ACVC1.11.C.E]X2120B. |
| \$NAME_SPECIFICATION3 | VMS386\$DKB0:[ACVC1.11.C.E]X3119A. |
| \$NEG_BASED_INT       | 16#F000000E#                       |
| \$NEW_MEM_SIZE        | 16_777_216                         |
| \$NEW_STOR_UNIT       | 8                                  |
| \$NEW_SYS_NAME        | I386                               |
| \$PAGE_TERMINATOR     | ASCII.FF                           |
| \$RECORD_DEFINITION   | RECORD SUBP: OPERAND; END RECORD;  |
| \$RECORD_NAME         | CODE_0                             |
| \$TASK_SIZE           | 32                                 |
| \$TASK_STORAGE_SIZE   | 1024                               |
| \$TICK                | 0.01                               |
| \$VARIABLE_ADDRESS    | VAR_3'ADDRESS                      |
| \$VARIABLE_ADDRESS1   | VAR_1'ADDRESS                      |
| \$VARIABLE_ADDRESS2   | VAR_2'ADDRESS                      |
| \$YOUR_PRAGMA         | PRAGMA PASSIVE                     |



APPENDIX B  
COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

## COMPILATION SYSTEM OPTIONS

VADS ADA - Ada compiler

### Syntax

VADS ADA source\_file [, ...]

### Qualifiers

/APPEND      Append all output to a log file.

/DEBUG

/DEBUG=G     Write out the gnrx.lib file in ASCII.

/DEFINE

/DEFINE=(identifier:type=value", ... )

Define identifier of a specified type and value. See VADS ADA PREPROCESSOR REFERENCE.

/DEPENDENCIES      Analyze for dependencies only; no link will be performed this option is given (/MAIN and /OUTPUT options must not be used with this qualifier).

/ERRORS

/ERRORS[=(option [, ...])] Process compilation error messages using the ERR tool and direct the output to SYSSOUTPUT; the parentheses can be omitted if only one qualifier is given (by default, only lines containing errors are listed).

Options:

LISTING      List entire input file.

EDITOR[="editor"]

Insert error messages into the source file and call a text editor (EDT by default). If a value is given as a quoted string, that string is used to invoke the editor. This allows other editors to be used instead of the default.

OUTPUT[=file\_name]

Direct error processed output to the specified file name; if no file name is given, the source file name is used with a file extension .ERR.

BRIEF          List only the affected lines [default]

Only one of the BRIEF, LISTING, OUTPUT, or EDITOR options can be used in a single command.

For more information about the /ERRORS option, see also COMPILING ADA PROGRAMS, COMPILER ERROR MESSAGE PROCESSING.

/EXECUTABLE

/EXECUTABLE=file name

Provide an explicit name for the executable when used with the /MAIN qualif the file\_name value must be supplied (if the file type is omitted,

## COMPILATION SYSTEM OPTIONS

.EXE is assumed).

/KEEP IL Keep the intermediate language (IL) file produced by the compiler front end. The IL file will be placed in the OBJECTS directory, with the name ADA\_SOURCE.I.

/LIBRARY

/LIBRARY=library\_name Operate in VADS

library library\_name (the current working directory is the default).

/LINK ARGUMENTS

/LINK ARGUMENTS="value"

Pass command qualifiers and parameters to the linker.

/MAIN

/MAIN[=unit\_name] Produce an executable program using the named unit as the main program; if no value is given, the name is derived from the first Ada file name parameter (the .A suffix is removed); the executable file name is derived from the main program name unless the /EXECUTABLE qualifier is used.

/NOOPTIMIZE Do not optimize.

/WARNINGS Print warning diagnostics.

/OPTIMIZE

/OPTIMIZE[=number] Invoke the code optimizer (OPTIM3). An optional digit provides the level of optimization. /OPTIMIZE=4 is the default.

/OPTIMIZE no digit, full optimization

/OPTIMIZE=0 prevents optimization

/OPTIMIZE=1 no hoisting

/OPTIMIZE=2 no hoisting, but more passes

/OPTIMIZE=3 no hoisting, but even more passes

/OPTIMIZE=4 hoisting from loops

/OPTIMIZE=5 hoisting from loops, but more passes

/OPTIMIZE=6 hoisting from loops with maximum passes

/OPTIMIZE=7 hoisting from loops and branches

/OPTIMIZE=8 hoisting from loops and branches, more passes

/OPTIMIZE=9 hoisting from loops and branches, maximum passes

Hoisting from branches (and cases alternatives) can be slow and does not always provide significant performance gains, so it can be suppressed.

For more information about optimization, see COMPILING ADA PROGRAMS, Optimization. See also pragma OPTIMIZE\_CODE(OFF).

/OUTPUT

/OUTPUT=file\_name Direct the output to file\_name (the default is SYSS\$OUTPUT).

/PRE PROCESS Invoke the Ada Preprocessor. See VADS ADA PREPROCESSOR REFERENCE.

## COMPILATION SYSTEM OPTIONS

`/RECOMPILE_LIBRARY`  
`/RECOMPILE_LIBRARY=VADS_library`  
Force analysis of all generic instantiations causing reinstantiation of any that are out of date.

`/SHOW` Show the name of the tool executable but do not execute it.

`/SHOW ALL` Print the name of the front end, code generator, optimizer, and linker, and list the tools that will be invoked.

`/SUPPRESS` Apply pragma `SUPPRESS` for all checks to the entire compilation. (See also pragma `SUPPRESS(ALL_CHECKS)`)

`/TIMING` Print timing information for the compilation.

`/VERBOSE` Print information for the compilation.

### Description

The command `VADS ADA` executes the Ada compiler and compiles the named Ada source file, ending with the `.A` suffix. The file must reside in a VADS library directory. The `ADA.LIB` file in this directory is modified after each Ada unit is compiled.

By default, `VADS ADA` produces only object and net files. If the `/MAIN` option is used, the compiler automatically invokes `VADS LD` and builds a complete program with the named library unit as the main program.

Non-Ada object files may be given as arguments to `VADS ADA`. These files will be passed on to the linker and will be linked with the specified Ada object files.

Command line options may be specified in any order, but the order of compilation and the order of the files to be passed to the linker can be significant.

Several VADS compilers may be simultaneously available on a single system. The `VADS ADA` command within any version of VADS on a system will execute the correct compiler components based upon visible library directives.

Program listings with a disassembly of machine code instructions are generated by `VADS DB` or `VADS DAS`.

### Diagnostics

The diagnostics produced by the VADS compiler are intended to be self-explanatory. Most refer to the RM. Each RM reference includes a section number and optionally, a paragraph number enclosed in parentheses.

## COMPILATION SYSTEM OPTIONS

### LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

VADS LD - prelinker

#### Syntax

VADS LD unit\_name

#### Qualifiers

/AF ND

/DEBUG Debug memory overflow (use in cases where linking a large number units causes the error message "local symbol overflow" to occur).

/EARLY

/EARLY="unit name" Force the given unit to be elaborated as early as possible (unit\_name must be enclosed in double quotes).

/EXECUTABLE

/EXECUTABLE[=file\_name]

Put the output in the named file. The default executable names are <main\_unit>.EXE on self-hosts or <main\_unit>.VOX on cross targets.

/FILES Print a list of dependent files in elaboration order and suppress linking.

/LIBRARY

/LIBRARY=library\_name Operate in VADS library library\_name (the current working directory is the default).

/LINK OPTIONS

/LINK OPTIONS=object\_file\_or\_qualifier [,...]"

Add the options surrounded by quotes to the invocation of the linker.

## COMPILATION SYSTEM OPTIONS

### /OUTPUT

/OUTPUT=file\_name Direct output to file\_name. Default is SYSS\$OUTPUT.

/SHOW Show the name of the tool executable but do not execute it.

/UNITS Print a list of dependent units in order and suppress linking.

/VERBOSE Print the VMS linker command prior to execution.

/VERIFY Print the VMS linker command but suppress execution.

### Description

VADS LD collects the object files needed to make unit\_name a main program and calls the VMS linker to link together all Ada and other language objects required to produce an executable. unit\_name must be a non-generic subprogram that is either a procedure or a function that returns an Ada STANDARD.INTEGER (the predefined type INTEGER). The utility uses the net files produced by the Ada compiler to check dependency information. VADS LD produces an exception mapping table, a unit elaboration table, and passes this information to the linker.

VADS LD reads instructions for generating executables from the ADA.LIB file in the VADS libraries on the search list. Besides information generated by the compiler, these directives also include WITHn directives that allow the automatic linking of object modules compiled from other languages or Ada object modules not named in context clauses in the Ada source. Any number of WITH directives may be placed into a library, but they must be numbered contiguously beginning at WITH1. The directives are recorded in the library's ADA.LIB file and have the following form:

```
WITH1|LINK|object_file|  
WITH2|LINK|archive_file|
```

WITH directives may be placed in the local Ada libraries or in any VADS library on the search list.

A WITH directive in a local VADS library or earlier on the library search list will hide the same numbered WITH directive in a library later in the library search list.

Use VADS INFO to change or report library directives in the current library.

All arguments after unit\_name are passed on to the linker. These arguments may be linker options, names of object files or archive libraries, or library abbreviations.

### Diagnostics

Self-explanatory diagnostics are produced for missing files, etc. Occasional additional messages are produced by the linker.

## COMPILATION SYSTEM OPTIONS

### Files

Normally VADS LD generates an intermediate file with the process ID as a substring, VADSOPTION<process\_ID>.OPT.

With either the /VERIFY or /VERBOSE qualifiers,; however, VADS LD will produce the intermediate file <main\_unit>.OPT.

ATTACHMENT I

APPENDIX F. Implementation-Dependent Characteristics

1. Implementation-Dependent Pragmas

1.1. INITIALIZE Pragma

Takes one of the identifiers `STATIC` or `DYNAMIC` as the single argument. This pragma is only allowed within a library-level package spec or body. It specifies that all objects in the package be initialized as requested by the pragma (i.e. statically or dynamically). Only library-level objects are subject to static initialization; all objects within procedures are always (by definition) dynamic. If pragma `INITIALIZE(STATIC)` is used, and an object cannot be initialized statically, code will be generated to initialize the object, and a warning message will be generated.

1.2. `INLINE_ONLY` Pragma

The `INLINE_ONLY` pragma, when used in the same way as pragma `INLINE`, indicates to the compiler that the subprogram must always be inlined. This pragma also suppresses the generation of a callable version of the routine which saves code space. If a user erroneously makes an `INLINE_ONLY` subprogram recursive, a warning message will be emitted and an `PROGRAM_ERROR` will be raised at run time.

1.3. `BUILT_IN` Pragma

The `BUILT_IN` pragma is used in the implementation of some predefined Ada packages, but provides no user access. It is used only to implement code bodies for which no actual Ada body can be provided, for example, the `MACHINE_CODE` package.

1.4. `SHARE_CODE` Pragma

The `SHARE_CODE` pragma takes the name of a generic instantiation or a generic unit as the first argument and one of the identifiers `TRUE` or `FALSE` as the second argument. This pragma is only allowed immediately at the place of a



declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

When the first argument is a generic unit, the pragma applies to all instantiations of that generic. When the first argument is the name of a generic instantiation, the pragma applies only to the specified instantiation, or overloaded instantiations.

If the second argument is TRUE, the compiler will try to share code generated for a generic instantiation with code generated for other instantiations of the same generic. When the second argument is FALSE, each instantiation will get a unique copy of the generated code. The extent to which code is shared between instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit.

The name pragma `SHARE_BODY` is also recognized by the implementation and has the same effect as `SHARE_CODE`. It is included for compatability with earlier versions of VADS.

#### 1.5. `NO_IMAGE` Pragma

The pragma suppresses the generation of the image array used for the `IMAGE` attribute of enumeration types. This eliminates the overhead required to store the array in the executable image. An attempt to use the `IMAGE` attribute on a type whose image array has been suppressed will result in a compilation warning and `PROGRAM_ERROR` raised at run time.

#### 1.6. `EXTERNAL_NAME` Pragma

The `EXTERNAL_NAME` pragma takes the name of a subprogram or variable defined in Ada and allows the user to specify a different external name that may be used to reference the entity from other languages. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification.

#### 1.7. `INTERFACE_NAME` Pragma

The `INTERFACE_NAME` pragma takes the name of a variable or subprogram defined in another language and allows it to be referenced directly in Ada. The pragma will replace all occurrences of the variable or subprogram name with an external reference to the second, link argument. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object or subprogram declared earlier in the same package specification. The

## APPENDIX F OF THE Ada STANDARD

object must be declared as a scalar or an access type. The object cannot be any of the following:

- a loop variable,
- a constant,
- an initialized variable,
- an array, or
- a record.

### 1.8. IMPLICIT\_CODE Pragma

Takes one of the identifiers ON or OFF as the single argument. This pragma is only allowed within a machine code procedure. It specifies that implicit code generated by the compiler be allowed or disallowed. A warning is issued if OFF is used and any implicit code needs to be generated. The default is ON.

### 1.9. OPTIMIZE\_CODE Pragma

Takes one of the identifiers ON or OFF as the single argument. This pragma is only allowed within a machine code procedure. It specifies whether the code should be optimized by the compiler. The default is ON. When OFF is specified, the compiler will generate the code as specified.

## 2. Implementation of Predefined Pragmas

### 2.1. CONTROLLED

This pragma is recognized by the implementation but has no effect.

### 2.2. ELABORATE

This pragma is implemented as described in Appendix B of the Ada RM.

### 2.3. INLINE

This pragma is implemented as described in Appendix B of the Ada RM.

### 2.4. INTERFACE

This pragma supports calls to 'C' and FORTRAN functions. The Ada subprograms can be either functions or procedures. The types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS in SYSTEM. All parameters must have mode IN. Record and array objects can be passed by reference using the ADDRESS attribute.

## 2.5. LIST

This pragma is implemented as described in Appendix B of the Ada RM.

## 2.6. MEMORY\_SIZE

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas; the SYSTEM package must be recompiled.

## 2.7. NON\_REENTRANT

This pragma takes one argument which can be the name of either a library subprogram or a subprogram declared immediately within a library package spec or body. It indicates to the compiler that the subprogram will not be called recursively, allowing the compiler to perform specific optimizations. The pragma can be applied to a subprogram or a set of overloaded subprograms within a package spec or package body.

## 2.8. NOT\_ELABORATED

This pragma can only appear in a library package specification. It indicates that the package will not be elaborated because it is either part of the RTS, a configuration package, or an Ada package that is referenced from a language other than Ada. The presence of this pragma suppresses the generation of elaboration code and issues warnings if elaboration code is required.

## 2.9. OPTIMIZE

This pragma is recognized by the implementation but has no effect.

## 2.10. PACK

This pragma will cause the compiler to choose a non-aligned representation for composite types. It will not cause objects to be packed at the bit level.

## 2.11. PAGE

This pragma is implemented as described in Appendix B of the Ada RM.

## 2.12. PASSIVE

## APPENDIX F OF THE Ada STANDARD

The pragma has three forms:

```
PRAGMA PASSIVE;  
PRAGMA PASSIVE(SEMAPHORE);  
PRAGMA PASSIVE(INTERRUPT, <number>);
```

This pragma Pragma passive can be applied to a task or task type declared immediately within a library package spec or body. The pragma directs the compiler to optimize certain tasking operations. It is possible that the statements in a task body will prevent the intended optimization; in these cases, a warning will be generated at compile time and will raise `TASKING_ERROR` at runtime.

### 2.13. PRIORITY

This pragma is implemented as described in Appendix B of the Ada RM.

### 2.14. SHARED

This pragma is recognized by the implementation but has no effect.

### 2.15. STORAGE\_UNIT

This pragma is recognized by the implementation. The implementation does not allow `SYSTEM` to be modified by means of pragmas; the `SYSTEM` package must be recompiled.

### 2.16. SUPPRESS

This pragma is implemented as described, except that `DIVISION_CHECK` and in some cases `OVERFLOW_CHECK` cannot be suppressed.

### 2.17. SYSTEM\_NAME

This pragma is recognized by the implementation. The implementation does not allow `SYSTEM` to be modified by means of pragmas; the `SYSTEM` package must be recompiled.

## 3. Implementation-Dependent Attributes

### 3.1. P'REF

For a prefix that denotes an object, a program unit, a label, or an entry:

This attribute denotes the effective address of the first of the storage units allocated to P. For a subprogram, pack-

age, task unit, or label, it refers to the address of the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, it refers to the corresponding hardware interrupt. The attribute is of the type OPERAND defined in the package MACHINE\_CODE. The attribute is only allowed within a machine code procedure.

See section F.4.8 for more information on the use of this attribute.

(For a package, task unit, or entry, the 'REF attribute is not supported.)

### 3.2. T'TASKID

For a task object or a value T, T'TASK ID yields the unique task id associated with a task. The value of this attribute is of the type ADDRESS in the package SYSTEM.

## 4. Specification of Package SYSTEM

-- Copyright 1987, 1988, 1989, 1990 Verdix Corporation

with unsigned types;  
package SYSTEM is

```
pragma suppress(ALL CHECKS);
pragma suppress(EXCEPTION_TABLES);
pragma not_elaborated;
```

```
type NAME is ( i386 );
```

```
SYSTEM_NAME          : constant NAME := i386;
```

```
STORAGE_UNIT        : constant := 8;
```

```
MEMORY_SIZE         : constant := 16_777_216;
```

```
-- System-Dependent Named Numbers
```

```
MIN_INT              : constant := -2_147_483_648;
```

```
MAX_INT              : constant := 2_147_483_647;
```

```
MAX_DIGITS           : constant := 15;
```

```
MAX_MANTISSA         : constant := 31;
```

```
FINE_DELTA           : constant := 2.0**(-31);
```

```
TICK                 : constant := 0.01;
```

```
-- Other System-dependent Declarations
```

## APPENDIX F OF THE Ada STANDARD

```
subtype PRIORITY is INTEGER range 0 .. 99;

MAX_REC_SIZE : integer := 64*1024;

type ADDRESS is private;

function ">" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
function "<" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
function ">=" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
function "<=" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
function "-" (A: ADDRESS; B: ADDRESS) return INTEGER;
function "+" (A: ADDRESS; I: INTEGER) return ADDRESS;
function "-" (A: ADDRESS; I: INTEGER) return ADDRESS;

function "+" (I: UNSIGNED_TYPES.UNSIGNED_INTEGER) return ADDRESS;

function MEMORY_ADDRESS
  (I: UNSIGNED_TYPES.UNSIGNED_INTEGER) return ADDRESS renames "+";

NO_ADDR : constant ADDRESS;

type TASK_ID is private;
NO_TASK_ID : constant TASK_ID;

type PROGRAM_ID is private;
NO_PROGRAM_ID : constant PROGRAM_ID;

private

type ADDRESS is new UNSIGNED_TYPES.UNSIGNED_INTEGER;

NO_ADDR : constant ADDRESS := 0;

pragma BUILT_IN(">");
pragma BUILT_IN("<");
pragma BUILT_IN(">=");
pragma BUILT_IN("<=");
pragma BUILT_IN("-");
pragma BUILT_IN("+");

type TASK_ID is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
NO_TASK_ID : constant TASK_ID := 0;

type PROGRAM_ID is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
NO_PROGRAM_ID : constant PROGRAM_ID := 0;

end SYSTEM;
```

### 5. Restrictions on Representation Clauses

### 5.1. Pragma PACK

In the absence of Pragma PACK, record components are padded so as to provide for efficient access by the target hardware; pragma PACK applied to a record eliminates the padding where possible. Pragma PACK has no other effect on the storage allocated for record components a record representation is required.

### 5.2. Size Clauses

For scalar types, a representation clause will pack to the number of bits required to represent the range of the subtype. A size clause applied to a record type will not cause packing of components; an explicit record representation clause must be given to specify the packing of the components. A size clause applied to a record type will cause packing of components only when the component type is a discrete type. An error will be issued if there is insufficient space allocated. The SIZE attribute is not supported for task, access, or floating point types.

### 5.3. Address Clauses

Address clauses are only supported for variables. Since default initialization of a variable requires evaluation of the variable address, elaboration ordering requirements prohibit initialization of variables which have address clauses. The specified address indicates the physical address associated with the variable.

### 5.4. Interrupts

Interrupt entries are not supported.

### 5.5. Representation Attributes

The ADDRESS attribute is not supported for the following entities:

- Packages
- Tasks
- Labels
- Entries

### 5.6. Machine Code Insertions

Machine code insertions are supported.

The general definition of the package MACHINE\_CODE provides

## APPENDIX F OF THE Ada STANDARD

an assembly language interface for the target machine. It provides the necessary record type(s) needed in the code statement, an enumeration type of all the opcode mnemonics, a set of register definitions, and a set of addressing mode functions.

The general syntax of a machine code statement is as follows:

```
CODE_n'( opcode, operand {, operand} );
```

where n indicates the number of operands in the aggregate.

A special case arises for a variable number of operands. The operands are listed within a subaggregate. The format is as follows:

```
CODE_N'( opcode, (operand {, operand}) );
```

For those opcodes that require no operands, named notation must be used (cf. RM 4.3(4)).

```
CODE_0'( op => opcode );
```

The opcode must be an enumeration literal (i.e., it cannot be an object, attribute, or a rename).

An operand can only be an entity defined in MACHINE\_CODE or the 'REF attribute.

The arguments to any of the functions defined in MACHINE\_CODE must be static expressions, string literals, or the functions defined in MACHINE\_CODE. The 'REF attribute may not be used as an argument in any of these functions.

Inline expansion of machine code procedures is supported.

### 6. Conventions for Implementation-generated Names

There are no implementation-generated names.

### 7. Interpretation of Expressions in Address Clauses

Address expressions in an address clause are interpreted as physical addresses.

### 8. Restrictions on Unchecked Conversions

None.

### 9. Restrictions on Unchecked Deallocations



None.

## 10. Implementation Characteristics of I/O Packages

Instantiations of `DIRECT IO` use the value `MAX_REC_SIZE` as the record size (expressed in `STORAGE_UNITS`) when the size of `ELEMENT_TYPE` exceeds that value. For example, for unconstrained arrays such as string, where `ELEMENT_TYPE'SIZE` is very large, `MAX_REC_SIZE` is used instead. `MAX_RECORD_SIZE` is defined in `SYSTEM` and can be changed by a program before instantiating `DIRECT IO` to provide an upper limit on the record size. In any case, the maximum size supported is  $1024 \times 1024 \times \text{STORAGE\_UNIT}$  bits. `DIRECT IO` will raise `USE_ERROR` if `MAX_REC_SIZE` exceeds this absolute limit.

Instantiations of `SEQUENTIAL IO` use the value `MAX_REC_SIZE` as the record size (expressed in `STORAGE_UNITS`) when the size of `ELEMENT_TYPE` exceeds that value. For example, for unconstrained arrays such as string, where `ELEMENT_TYPE'SIZE` is very large, `MAX_REC_SIZE` is used instead. `MAX_RECORD_SIZE` is defined in `SYSTEM` and can be changed by a program before instantiating `INTEGER IO` to provide an upper limit on the record size. `SEQUENTIAL IO` imposes no limit on `MAX_REC_SIZE`.

## 11. Implementation Limits

The following limits are actually enforced by the implementation. It is not intended to imply that resources up to or even near these limits are available to every program.

### 11.1. Line Length

The implementation supports a maximum line length of 500 characters including the end of line character.

### 11.2. Record and Array Sizes

The maximum size of a statically sized array type is  $4,000,000 \times \text{STORAGE\_UNITS}$ . The maximum size of a statically sized record type is  $4,000,000 \times \text{STORAGE\_UNITS}$ . A record type or array type declaration that exceeds these limits will generate a warning message.

### 11.3. Default Stack Size for Tasks

In the absence of an explicit `STORAGE_SIZE` length specification, every task except the main program is allocated a fixed size stack of 10,240 `STORAGE_UNITS`. This is the value returned by `T'STORAGE_SIZE` for a task type `T`.

### 11.4. Default Collection Size

## APPENDIX F OF THE Ada STANDARD

In the absence of an explicit `STORAGE_SIZE` length attribute, the default collection size for an access type is 100 times the size of the designated type. This is the value returned by `T'SORAGE_SIZE` for an access type `T`.

### 11.5. Limit on Declared Objects

There is an absolute limit of  $6,000,000 \times \text{STORAGE\_UNITS}$  for objects declared statically within a compilation unit. If this value is exceeded, the compiler will terminate the compilation of the unit with a FATAL error message.